Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 216 (2008) 65–77
www.elsevier.com/locate/entcs

Declarative Debugging Meets the World

Wolfgang Lux[1]

Abstract

Declarative debugging has been proposed as a suitable technique for diagnosing errors. It is particularly attractive for declarative programming languages, whose operational semantics differs substantially from their declarative semantics. Declarative debuggers are usually described and implemented by means of a program transformation. However, this transformation does not mix well with monadic I/O, which is used by lazy functional and functional logic languages. Therefore, declarative debuggers for such languages either do not support debugging of programs involving stateful computations at all, or require special support from the compiler and its runtime system. In this paper, we present a modified program transformation that blends nicely with monadic I/O and therefore covers the whole language without additional support from the target implementation.

Keywords: declarative debugging, monadic I/O

1 Introduction

Declarative debugging has been proposed as a suitable technique for diagnosing errors on the basis of the intended meaning of a program. Since it abstracts from the operational behavior of programs, it is particularly attractive for debugging programs written in declarative programming languages. Understanding the operational behavior of programs implemented in such languages can become quite difficult due to lazy evaluation, concurrency, and non-deterministic search and thus distracts from finding the actual causes of an error.

The key idea of declarative debugging is to build a computation tree (also called an evaluation dependency tree) representing the computations performed by an erroneous program. The exact form of the computation tree depends on the kinds of errors that are to be diagnosed. Each node of a computation tree represents the outcome of a computation step in the program and is connected to the nodes representing the subcomputations that were performed in order to produce the result of the computation.

[1] Email: wlux@uni-muenster.de

doi:10.1016/j.entcs.2008.06.034

If a program does not compute the expected result or produces unexpected results, its computation tree contains one or more erroneous nodes, whose result does not match the semantics intended by the programmer. Some of these erroneous nodes are just the result of other erroneous computations, while other erroneous nodes represent genuine programming errors. These *buggy nodes* are distinguished by the fact that they do not have any erroneous children, i.e., their represented computation steps compute a wrong result from correct inputs.

Since a debugging session starts with an observation that the program does not work as expected, i.e., the root of the computation tree is an erroneous node, and all nodes of the computation tree are connected with that root, it is possible to find a buggy node by a top-down navigation of the computation tree. To that end, an *oracle* – usually the user – is asked questions about the computation steps recorded in the computation tree and whether they agree with the program's intended semantics or not.

A difficult aspect for implementing declarative debuggers for lazy functional and functional logic languages like Haskell [8], Curry [4], and $\mathcal{TOY}$ [5] is the integration of input and output. This difficulty is twofold. First, the program's own interaction with the external world must be kept separate from the interaction of the debugger with the oracle. Second, input and output is encapsulated in the `IO` monad, which has an implicit higher-order nature and is not directly eligible for a program transformation. For that reason, present declarative debuggers for these languages either do not cover I/O at all or use a substitute `IO` monad that must be mapped back to the real `IO` monad by means of an additional language primitive.

In this paper, we present a modification to the standard program transformation approach that avoids these shortcomings by transforming functions in the `IO` monad in a specific way. This is made feasible by the fact that `IO` is an abstract type and elements of that type cannot be inspected directly by user-defined functions. Therefore it is sufficient to define appropriate adaptors for the primitive `IO` functions. For almost all of these adaptor functions, their definition can be derived mechanically from the semantic model of the `IO` type. Only for a few distinguished primitives one needs to provide predefined implementations. Nevertheless, no additional non-standard primitives are needed for their definition. Thus, while using the semantic model of the type `IO` in order to derive the adaptors, their concrete implementation does not rely on any particular implementation technique of I/O actions in the target implementation.

The rest of this paper is structured as follows. In the next section, we briefly review the standard program transformation approach for detecting wrong answers in functional and functional logic programming languages. The third section describes monadic I/O and the problems that it poses for the program transformation approach. In section 4, we introduce our modified program transformation that mixes well with monadic I/O. Finally, the two last sections present related work and conclude.

```
reverse :: [a] → [a]
reverse []      = []
reverse (x:xs)  = append xs [x]

append :: [a] → [a] → [a]
append []      ys = ys
append (x:xs)  ys = x :  append xs ys
```

Fig. 1. A buggy reverse function

2 Declarative Debugging

In this section we introduce declarative debugging of wrong answers for lazy functional and functional logic languages informally. See [1,3] for a complete theoretical treatment. For our presentation, we will use the multi-paradigm declarative language Curry [4]. However, we do not rely on any specific features of Curry so the examples are equally valid (after minor syntactic changes) for $\mathcal{TOY}$ and – except where noted otherwise – for the functional language Haskell [8].

2.1 Computation Trees

The debugging process is started with an initial symptom detected while executing a goal. Two different kinds of symptoms are possible, corresponding to different kinds of bugs.

- A *positive symptom* is an unexpected answer obtained for the goal. This is called a *wrong answer*.
- A *negative symptom* is an expected answer that is missing from the multiset of results computed for the goal. This is a *missing answer*.

Often, missing answers and wrong answers occur together. For instance, consider the buggy program from Fig. 1. For this program, the goal `reverse [1,2,3]` yields the result `[2,3,1]`, which is a wrong answer. The expected result `[3,2,1]` is not computed, so there is a missing answer too.

In order to detect wrong answers, the oracle must know the intended meaning $\mathcal{I}$ of the program. As proved in [1], it is sufficient to consider $\mathcal{I}$ as a set of *basic facts* of the form $f\,t_1 \ldots t_n \to t$, where $t_1, \ldots, t_n, t$ are constructor terms. A basic fact $f\,t_1 \ldots t_n \to t$ means that function f produces the result t when applied to the arguments $t_1, \ldots, t_n$. The oracle is asked only questions whether a basic fact is in $\mathcal{I}$ or not. This ensures that the questions asked are as simple as possible and is achieved by replacing nested function applications by their results obtained during the computation. Applications that were not evaluated by the program are denoted by the special constructor term $\bot$, which is represented by the symbol `_` in the debugger's questions.

The *computation trees* used by the debugger will have basic facts at their nodes. Each node has an associated program rule, the program rule used at the corresponding computation step. Thus, the debugger will point out the program rule associated with a buggy node as an incorrect program rule. The children of a node correspond to the subcomputations carried out while evaluating the guard and right hand side of the program rule. The soundness and completeness results in [1] ensure that, given a wrong answer, an incorrect program rule is detected by the debugger.

2.2 Program Transformation

Several strategies have been proposed to create and navigate computation trees. A well-known approach widely employed in Logic Programming uses meta-interpreters to re-execute the goal during the debugging phase. Thus, the computation tree is not computed explicitly, and both wrong an missing answers are easily handled. This idea has been extended in the case of NUE-Prolog to functional logic languages [7].

However, this solution is not available to languages that do not provide built-in meta-instructions, as Haskell or Curry. To the best of our knowledge, only declarative debuggers for *wrong answers* have been developed for functional and functional logic languages. The reason is twofold: first, the computation trees necessary for detecting missing answers are much more complicated. Second, often wrong and missing answers occur simultaneously, as in the case of our buggy `reverse` example. In these cases, it is enough to find out the reason for the wrong answer to get rid of both errors. This is also the case of our debugger.

Two different techniques have been proposed in related papers for producing a computation tree associated with a wrong answer (see [6] for a comparison):

(i) Modify the implementation of the abstract machine to produce the computation tree during the goal's evaluation.

(ii) Transform the source program P into a new program P' in which all functions return the same result as in P, but paired with their corresponding computation tree.

We have adopted the latter approach because of its greater flexibility and portability. Computation trees are represented in transformed programs by elements of the type

```
data CTree = Void | Node Rule [Term] Term [CTree]
```

where `Rule` and `Term` are types suitable for identifying program rules and representing constructor terms, respectively. For simplicity, we will assume throughout the rest of this paper that `Rule` is just an alias for the type `String` and identify program rules simply by their function name. The `Term` type must be capable of representing unevaluated expressions. However, it is sufficient to use the same representation for all unevaluated expressions. For instance,

```
data Term = CAp String [Term] | Var String | Bottom
```

would be a suitable (untyped) representation, with `CAp` $c\,[t_1,\ldots,t_n]$ representing an application of constructor c to the argument terms $t_1,\ldots,t_n$, `Var` x an unbound

logical variable x, and `Bottom` an unevaluated expression.

`Void` nodes in computation trees are used for transformed functions which cannot be the cause of an error (*trusted functions*). A `Node` f `[`$t_1, \ldots, t_n$`]` t *cts* represents a basic fact $f\,t_1 \ldots t_n \to t$ with subcomputations represented by the elements of the list *cts*. In order to transform – possibly unevaluated – expressions into terms, we use an impure function `dval :: a → Term` that converts its argument without further evaluation. The result of `dval` e therefore depends on the order of evaluation. However, this dependency on evaluation order is unproblematic because the computation tree is traversed only after the final result of the program has been computed. In fact, `dval`'s impureness is essential because the debugger must cause no further evaluations to the (sub)expressions of the original program while navigating the computation tree.

The idea of the program transformation is to transform each n-ary function $f :: \tau_1 \to \ldots \to \tau_n \to \tau$ of the program into a transformed function

$$f' :: \tau_1' \to \ldots \to \tau_n' \to (\tau', \texttt{CTree})$$

where the transformation of types $(\cdot)'$ is defined by the following rules:

$$\begin{aligned} \alpha' &= \alpha \\ (T\,\overline{\tau_n})' &= T\,\overline{\tau_n'} \\ (\mu \to \nu)' &= \mu' \to (\nu', \texttt{CTree}) \end{aligned} \tag{1}$$

Here, α is type variable, T is a type constructor of arity n, and μ and ν are types. Note that the transformation does not presume that functions are η-expanded, i.e., the result type τ of a function may contain arrows. However, the transformed types of η-expanded and η-reduced functions differ. For instance, the `append` function from Fig. 1 is transformed into a function

```
append' :: [a] → [a] → ([a],CTree)
```

whereas for the following, somewhat unusual definition of `append`

```
append :: [a] → [a] → [a]
append []       = \ys → ys
append (x:xs)   = \ys → x :  append xs ys
```

the transformed function would have type

```
append' :: [a] → ([a] → ([a],CTree), CTree)
```

By the transformation (1), only partial applications lacking a single argument have the correct signature to be used as arguments of higher-order functions. Therefore, auxiliary functions are introduced by the program transformation in order to adapt partial applications with two or more missing arguments to their expected types. We will not consider these auxiliary functions further in this paper. The type transformation $(\cdot)'$ is also applied to the argument types of all data constructors. However, the left-hand sides of the corresponding type declarations are unaffected

by the transformation.

The source code of a function is transformed by adding a computation tree to the result and flattening nested applications into simple applications using auxiliary local declarations. The details of the transformation can be found in [3]. As an example, the second equation of the `append` function from Fig. 1 is transformed into (an equivalent of) the following declaration.

```
append' (x:xs) ys = (r,t)
  where  (r1,t1) = append' xs ys
         r = x :  r1
         t = Node "append"  [dval (x:xs), dval ys] (dval r)
                            (clean [(r1,t1)])
```

For simplicity, we have assumed that rules are represented by their names. The auxiliary function

```
clean :: [(Term,CTree)] → [CTree]
clean cts = [t | (r,t) ← cts, dval r /= Bottom]
```

prunes the list of subcomputation trees by removing all trees which are associated with an unevaluated expression. This is necessary in order to ensure that computation trees remain finite in the presence of potentially infinite computations, thereby making an efficient depth-first traversal of computation trees in the debugger feasible.

Trivial wrappers are introduced as transformed functions for primitives, e.g., the arithmetic operations on integer numbers. These wrappers simply pair the result of the primitive with a `Void` node. For instance, the following auxiliary function is provided for integer addition

```
add_int' x y = (x + y,Void)
```

3 Debugging Interactive Programs

Monadic I/O has been introduced by Wadler in a series of papers [12,13], and since then has become popular for introducing stateful computations, and in particular input and output operations, into pure functional and functional logic languages with a non-strict semantics.

Programs interacting with the external world have type `IO` t, which can be considered an abbreviation for

$$\texttt{IO}\ t = \mathit{World} \to (t, \mathit{World}) \tag{2}$$

where *World* is a representation of the state manipulated by the program. However, the type `IO` is abstract in order to ensure a disciplined use of the state parameter. In particular, this prevents user programs from accidentally duplicating the state. In the following, we will use Eq. (2) as a semantic model of type `IO`. However, this does

not imply that implementations are restricted to implement type `IO` as a function of that particular type.

The two fundamental operations for type `IO` are

```
return :: a → IO a

(>>=) :: IO a → (a → IO b) → IO b
```

An expression `return` e lifts the result of expression e into the `IO` monad. An expression m `>>=` f first performs the action associated with the expression m and then the action associated with the expression $f\,x$, where x is the result delivered by the first action. Using our semantic model (2) for type `IO`, these functions can be defined as follows

```
return x = \world → (x,world)

m >>= f = \world → let (x,world') = m world in f x world'
```
(3)

Notice that `return` does not change the state and how `(>>=)` carefully ensures a single threaded use of the state.

The interaction with the external world is implemented by primitive operations. For instance, Haskell and Curry programs can use the primitive functions `getLine :: IO String` and `putStrLn :: String → IO ()` which implement actions that read a line from the standard input and write a line to the standard output, respectively.

Monadic I/O presents two challenges for a declarative debugger. First, one must carefully separate the debugged program's interaction with the external world and the debugger's interaction with the oracle. Second, one must provide appropriate computation trees for the primitive `IO` functions. The first of these problems can be solved easily by executing the program to completion before starting the debugger's navigation of the computation tree. The second problem is more difficult.

As noted above, for each primitive function the standard program transformation introduces an auxiliary function, which pairs the result of the primitive with a `Void` node. However, this approach is valid only for first-order primitives and not for functions like `(>>=)`. This is witnessed by `(>>=)`'s type after applying the transformation (1):[2]

```
(>>=') :: IO a → (a → (IO b,CTree)) → (IO b,CTree)
```

In a purely functional language, there is no safe way to implement a (useful) function with that type because the result of the first argument's I/O action, to which the second argument is supposed to be applied, is available only after execution of that action. This execution can take place only in an `IO` computation, but a function with the type given above is a pure function and not an `IO` computation.

In a functional logic language like Curry one might consider using logical variables in order to implement an auxiliary function for `(>>=)`. In a spirit similar to

[2] The operator symbol `(>>=')` is not legal in neither Haskell nor Curry. Nevertheless, we use this symbol in this paper in order to emphasize the relation between the `(>>=)` primitive and its auxiliary function.

our approach to extend the program transformation approach to cover encapsulated search [2], one might come up with the following definition.

```
m >>=' f =
  ((m >>= \x → let (m',t') = f x in t=:=t' &> m'), t)
  where t free
```

Here, the expression e_1 `=:=` e_2 denotes a strict equality on data terms between the two expressions e_1 and e_2 instantiating unbound variables in the two expression as necessary. The predefined operator `(&>)` implements a restricted identity such that c `&>` e is equivalent to e if the constraint c is satisfied, and fails otherwise.

Unfortunately, it turns out that the above definition for `(>>=')` does not work in general because the free variable `t` is instantiated too early by the equality constraint, namely before the monadic action `m'` has been executed at all. Thus, if the result of the subexpression `f x` is computed by an application of `(>>=')`, the variable `t'` is an unbound variable itself. In the end, this may lead to either an instantiation error or an unintended non-deterministic search in the `clean` function.

4 An Improved Program Transformation

The discussion in the previous section shows that the standard transformation cannot be used directly for programs using monadic I/O. In order to extend our transformation to monadic I/O, we will now have a more careful look at the program transformation.

If `IO` were not an abstract type, but defined as in Eq. (2), we could apply our standard transformation rules. In that case, the type `IO` τ would be equivalent to $\mathit{World} \to (\tau, \mathit{World})$ and would be transformed into

$$\mathit{World} \to ((\tau', \mathit{World}), \texttt{CTree}).$$

Obviously, this type is not an instance of `IO`, which is the source of our difficulties with monadic I/O. However, there is an (almost) isomorphic type[3], which is an instance of `IO`, namely

$$\mathit{World} \to ((\tau', \texttt{CTree}), \mathit{World}) = \texttt{IO}\ (\tau', \texttt{CTree}).$$

This isomorphism suggests a refinement of the type transformation (1), introducing a special case for type `IO`:

$$\begin{aligned} \alpha' &= \alpha \\ (\texttt{IO}\,\tau)' &= \texttt{IO}\ (\tau', \texttt{CTree}) \\ (T\,\overline{\tau_n})' &= T\,\overline{\tau_n'} \qquad\qquad (T \neq \texttt{IO}) \\ (\mu \to \nu)' &= \mu' \to (\nu', \texttt{CTree}) \end{aligned} \tag{4}$$

[3] Curry, like Haskell, has lifted products, i.e., $(\bot, \bot) \neq \bot$. Thus, in principle there is an observable difference between expressions of type $((\tau, \texttt{CTree}), \mathit{World})$ and expressions of type $((\tau, \mathit{World}), \texttt{CTree})$. However, this difference is not relevant here because we are concerned only with terminating programs and for those programs the inner pairs will never be equal to $\bot$.

Since the type `IO` is abstract and therefore user-defined functions cannot inspect elements of that type, the source code transformation for them is unaffected by the special treatment of type `IO`. In conjunction with the fact that the types $((\tau, \texttt{CTree}), \mathit{World})$ and $((\tau, \mathit{World}), \texttt{CTree})$ are equivalent as far as the debugger is concerned, this also ensures that the soundness and completeness results from [1] are valid for our modified transformation, too, provided that we can define suitable auxiliary functions for the primitive I/O operations.

4.1 Return

Things are very simple for `return`. If `return` were a user-defined function as in (3), it would be transformed into

```
return' x w = ((x,w), Void)
```

Flipping the tuple elements in the result, this becomes

```
return' x w = ((x,Void), w)
```

which is equivalent to `return' x = return (x,Void)`. However, as we noted in Sect. 2, η-reduction also changes the type of the transformed function and we end up with the following definition for our auxiliary function.

```
return' :: a → (IO (a, CTree), CTree)

return' x = (return (x, Void), Void)
```

Notice how the presence of two computation trees in the auxiliary function nicely reflects the double nature of the `IO` monad as pure computations that deliver another, stateful computation.

4.2 Sequential Execution

In order to provide an auxiliary function for the `(>>=)` primitive, we also consider its implementation as a user-defined function first. Its definition from (3) would be transformed into an equation

```
(>>=') m f w = ((r,w2), t)
  where  ((x,w1),t1)  = m w
         (m',t2)      = f x
         ((r,w2),t3)  = m' w2
         t = Node  "(>>=)" [dval m,dval f,dval w] (dval (r,w2))
                   [t1,t2,t3]
```

Note that we do not apply `clean` to the list of computation trees `[t1,t2,t3]` because we know that their associated expressions are evaluated to head normal form.

By transforming all expressions with type $((\tau, \mathit{World}), \texttt{CTree})$ into expressions of type $((\tau, \texttt{CTree}), \mathit{World})$, this definition becomes

```
(>>=') m f w = ((r, t),w2)
  where  ((x,t1),w1)  = m w
         (m',t2)      = f x
         ((r,t3),w2)  = m' w2
         t = Node  "(>>=)" [dval m,dval f,dval w] (dval (r,w2))
                   [t1,t2,t3]
```

It is straightforward to derive the following auxiliary function for `(>>=')` from this definition.

```
(>>=') :: IO (a,CTree) → (a → (IO (b,CTree), CTree))
       → (IO (b,CTree), CTree)
m >>=' f =
  ((m >>=                  \(x,t1) →
    let (m',t2) = f x in
    m' >>=                 \(r,t3) →
    let t = Node "(>>=)" [dval m,dval f] (dval r) [t1,t2,t3]),
   Void)
```

4.3 Other Primitives

Almost all other primitive I/O operations have a type of the form $\tau_1 \to \ldots \tau_n \to$ `IO` τ, where none of the τ_i nor τ is an arrow or `IO` type. According to our modified type transformation (4), we must provide an auxiliary function with type $\tau_1 \to \ldots \tau_n \to (\texttt{IO}\ (\tau, \texttt{CTree}), \texttt{CTree})$. The signature of these I/O operations indicates that operations are atomic with respect to the semantics of the host language. Therefore, both computation trees supplied by the auxiliary function can be only `Void` nodes. Thus, we introduce an auxiliary function

$$f'\, x_1 \ldots x_n = ((f\, x_1 \ldots x_n \texttt{ >>= } \backslash x \to (x, \texttt{Void})), \texttt{Void})$$

for each n-ary primitive with an `IO` result and no `IO` or function arguments. Note that the Haskell foreign function interface, which is also adopted by the Münster Curry compiler, allows defining foreign functions with an `IO` result, but does not allow `IO` or function arguments, so the above rule can be applied to such foreign declarations as well.

Definitions for the remaining primitives that have `IO` arguments or arguments with function types are more complicated. Fortunately, there are only a few of these primitives and the transformations can be derived from the semantic model,

too. In case of the Münster Curry compiler, there are exactly four primitives which require special treatment in addition to `(>>=)`: `unsafePerformIO`, `fixIO`, `catch`, and `encapsulate`.

An interesting case is the function `unsafePerformIO`. While its use is generally not recommended, it turns out to be useful in order to embed stateful computations into pure code, provided that one can prove that the resulting code does not break the declarative reading of the program. For instance, the purely functional `Array` library of the Münster Curry compiler is implemented on top of mutable arrays defined in the `IO` monad via a few safe uses of `unsafePerformIO`.

Using the semantic model (2), `unsafePerformIO` could be defined as follows

```
unsafePerformIO m = let (x,w') = m w in x
```

where `w` is a global constant that acts as substitute for the state of the external world while executing the monadic action. Applying the standard transformation to this definition and converting $((\tau, \mathit{World}), \texttt{CTree})$ expressions into $((\tau, \texttt{CTree}), \mathit{World})$, we arrive at the following definition of an auxiliary function for `unsafePerformIO`.

```
unsafePerformIO' m = (x,t)
  where  ((x,t),w') = m w
```

Yet, this is nothing other than `unsafePerformIO` itself.

In order to derive a transformed implementation of the `catch` primitive, which allows catching exceptions in monadic code, one must extend the semantic model (2) to allow for exceptions.

$$\texttt{IO}\ t = \mathit{World} \to (\texttt{Either IOError}\ t, \mathit{World}) \tag{5}$$

where `IOError` is an (abstract) representation of exceptions. The definitions of the primitives `return`, `(>>=)`, `catch`, and `fail`, which raises an exception, then become

```
return x = \w → (Right x,w)

m >>= f = \w → let (r,w') = m w in case r of
                 Left e → (e,w')
                 Right x → f x w'

fail e = \w → (Left e,w)

catch m f = \w → let (r,w') = m w in
                   Left e → f e w'
                   Right x → (x,w')
```

An important point is that for the extended semantic model the transformed type

$$(\texttt{IO}\ \tau)' = \mathit{World} \to ((\texttt{Either IOError}\ \tau', \mathit{World}), \texttt{CTree}).$$

is no longer isomorphic to

$$\texttt{IO}\ (\tau', \texttt{CTree}) = \mathit{World} \to ((\texttt{Either IOError}\ (\tau', \texttt{CTree}), \mathit{World}).$$

By transforming expressions according to

$$((\texttt{Left}\ e, w), t) \rightsquigarrow (\texttt{Left}\ e, w)$$

$$((\texttt{Right}\ x, w), t) \rightsquigarrow ((\texttt{Right}\ x, t), w)$$

we loose the computation trees in the first case. However, this is unproblematic. Recall that we are diagnosing only *wrong* answers. Yet, the error case $(\texttt{Left}\ e, w)$ corresponds to an exception condition where no answer is computed. Keeping the computation trees of such computations therefore is not necessary.

5 Related Work

Declarative debugging was introduced by Shapiro as a technique to diagnose errors in Prolog programs in [11]. This technique was later generalized into a generic debugging scheme by Naish [6].

The standard program transformation for functional logic programs, on which our transformation is based, was introduced and proven sound in [1,3]. In [2] it was shown that the transformation can be applied to Curry programs using encapsulated search with the help of a suitable auxiliary function.

Buddha [9] is a declarative debugger for Haskell and is based on a program transformation similar to ours [10]. I/O is handled by introducing a substitute monad `MIO`, which features an explicit $\mathit{World} \to (\tau, \mathit{World})$ definition and thus makes `IO` functions eligible to program transformation. However, this approach requires the introduction of a new primitive function `unsafeSyncIO` in order to embed the substitute in the real `IO` monad, which means that the soundness and correctness proofs from [1,3] no longer apply.

6 Conclusion

In this paper we have presented a modified program transformation approach for declarative debugging that mixes well with monadic I/O. Using a modified transformation rule for type `IO`, we avoid the need to disclose the implementation of type `IO` and do not need to resort to controversial functions like `unsafePerformIO` and `seq`. In conjunction with our previous work on declarative debugging and encapsulated search this now gives a full account to declarative debugging of Curry programs.

A prototype implementation of the debugger is part of the Münster Curry compiler [4] and can be used for experimenting with the debugger.

The present implementation of the debugger executes the whole program to completion before entering the navigation phase. While easy to understand from the perspective of the user and straightforward to implement, this approach suffers from the high memory demands of the transformed programs, which essentially keep a trace of the whole program's execution in memory. We plan to investigate

[4] http://danae.uni-muenster.de/~lux/curry

improving the debugger in this respect by writing our partial traces of the program's execution to disk and eventually interleaving the navigation phase with the program's execution. The difficult aspect here is to reconcile the two conflicting demands to write out program traces early in order to reduce memory usage on one hand and to provide sufficient information for the user to answer the debugger's questions on the other hand.

References

[1] Caballero, R., F. J. López-Fraguas and M. Rodríguez-Artalejo, *Theoretical foundations for the declarative debugging of lazy functional logic programs*, in: H. Kuchen and K. Ueda, editors, *Proc. FLOPS 2001*, LNCS 2024 (2001), pp. 170–184.

[2] Caballero, R. and W. Lux, *Declarative debugging for encapsulated search* **76** (2002).
URL http://www.elsevier.com/locate/entcs/volume76.html

[3] Caballero, R. and M. Rodríguez-Artalejo, *A declarative debugging system for lazy functional logic programs*, Electronic Notes in Theoretical Computer Science **64** (2002).

[4] Hanus (ed.), M., *Curry: An integrated functional logic language. (version 0.8.2)*, (2006).
URL http://www.informatik.uni-kiel.de/~mh/curry/report.html

[5] López-Fraguas, F. J. and J. Sánchez-Hernández, $\mathcal{TOY}$ *a multiparadigm declarative system*, in: M. R. Paliath Narendran, editor, *Proc. RTA-99*, LNCS 1631 (1999), pp. 244–247.

[6] Naish, L., *A declarative debugging scheme*, Journal of Functional and Logic Programming **3** (1997).

[7] Naish, L. and T. Barbour, *A declarative debugger for a logical-functional language*, in: G. Forsyth and M. Ali, editors, *Eight International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Invited and additional papers*, 1995, pp. 91–99.

[8] Peyton Jones, S. L., editor, "Haskell 98 Language and Libraries The Revised Report," Cambridge University Press, 2003. URL http://haskell.org/onlinereport/

[9] Pope, B., *Declarative debugging with Buddha*, in: V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, LNCS **3622** (2005), pp. 273–308.

[10] Pope, B. and L. Naish, *Practical aspects of declarative debugging in Haskell-98*, in: *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, 2003, pp. 230–240.

[11] Shapiro, E. Y., *Algorithmic program debugging*, in: *Proc. POPL'82* (1982), pp. 299–308.

[12] Wadler, P., *Comprehending monads*, Mathematical Structures in Computer Science **2** (1992), pp. 461–493.

[13] Wadler, P., *The essence of functional programming*, in: *Proc. POPL'92*, 1992, pp. 1–14.